Some Improvements to the Abstract Syntax for Analysis and Optimization of Full (Parallel) Prolog

F. Bueno
D. Cabeza
M. Hermenegildo
S. Prestwich
G. Puebla

1 Introduction

In deliverable D.WP1.1.M1 [1] of ESPRIT project #6707 *ParForce*, we proposed an abstract syntax for Prolog that will help the manipulation of programs at compile–time, as well as the exchange of sources and information among the tools designed for this manipulation, and between the programmer and the tools. We concentrated there on the information exchange format, rather than on the syntax of programs, for which we assumed a simplified format. Our purpose was to provide a low–level meeting point for the tools which will allow them to read the same programs and understand the information about them.

In this report we present some syntactical and semantical improvements to the work presented then. We discuss some source-to-source transformation techniques for *pruning* predicates, as well as new declarations and modifications to previous ones to deal with meta-calls, dynamic predicates, and multiple program specialization. We also show some analysis techniques that allow overcoming the difficulties in analysing *full* Prolog programs.

2 Pruning predicates — remote cuts

In the original report we adopted a low–level "plain" interchange syntax based on Edinburgh Prolog, which we said could be obtained by a simple preprocessing of the programs to remove complex constructions as disjunctions or if-then-elses. The idea was to have a sufficiently simple syntax for the interchange among tools, disregarding any "syntactic sugar" which could be added to it. But in the transformation of cuts inside disjunctions (or equivalently, if-then-elses) into this plain syntax a subtle problem arises, not addressed there.

The problem is that such cuts refer to the whole predicate where they appear, and the transformation to the interchange syntax creates new predicates out of disjunctions or if-then-elses, therefore changing the scope of the pruning.

For example, given this Prolog predicate with disjunctions

```
p :- a, (b, !, c ; d).
P :- e.
```

a naïve transformation in order to remove the disjunction would be

```
p :- a, p_disj.
p :- e.
```

```
p_disj :- b, !, c.
p_disj :- d.
```

However, clearly the resulting program is not equivalent to the original one, since the cut no longer prunes the alternatives of "a" nor the other alternatives of "p".

Thus, we need a mechanism to explicitly mark the scope of the cut (a "remote" cut). In order to do this we adopt the same solution used in many traditional Prolog systems, such as SICStus Prolog [3]. One way of doing it is as follows:

```
p :- get_choice(B), a, p_disj(B).
p :- e.

p_disj(B) :- b, cut(B), c.
p_disj(_) :- d.
```

where `get_choice(B)` unifies `B` with the current choice point and `cut(B)` cuts the tree up to `B` (in the example, the choice point for `p`).

Alternatively, `get_choice(B)` can be defined to unify `B` with the next choice point. In that case, the primitive must be called before entering the predicate. For example:

```
p:- get_choice(B), pp(B).

pp(B) :- a, p_disj(B).
pp(B) :- e.

p_disj(B) :- b, cut(B), c.
p_disj(_) :- d.
```

We are designing our tools in such a way that can deal with both solutions, depending on the particular implementation of "remote cuts" in the target implementation.

3 Meta-calls

Abstract interpretation tries to determine at compile-time (a superset of) all the possible values that variables in the program can take at run-time. Logic programs have a well established semantics that makes them good candidates for abstract interpretation. However, full Prolog programs have some meta– and extra–logical characteristics that apparently make it difficult or even impossible to analyse them accurately.

In D.WP1.1M1 we showed how a simple program transformation could be used to analyse and optimize programs that contain meta-calls. In this addendum we treat meta-calls again, first in order to further clarify the ideas introduced in the previous deliverable and second because treatment of meta-calls is crucial for dynamic predicates, as we will see in section 4.

An important characteristic of logic programs is that both programs and data can be represented as logical terms. Meta-calls are literals in a program in which one of the arguments will be called (executed) at run-time. Thus, a term is converted into a goal. Depending on the degree of instantiation at compile-time of the term that will be called at run-time, meta-calls will be more or less difficult to manage. This is because meta-calls may generate calls to predicates that are not determined at compile-time. If the term, as it appears in the program, is completely ground, then meta-calls can be analysed without problems. If the term is partially instantiated and we can determine the main functor, either from the program or as a result of the analysis (e.g. p(X,Y)), then we know the predicate that will be called at run-time. We only have to analyse the predicate *p/2* with the current abstract substitution for X and Y. The last and worst case is when the term to be called is just a variable at compile-time. We will call these *indeterminate* meta-calls[1]. In this case, any of the predicates in the program may be called, and what is worse, with any kind of substitution. This means that all the predicates in the program must be prepared to receive any input value. Thus they cannot be specialized with respect to any set of input values, and the only correct thing to do is to keep the original program.

The program transformation mentioned above implies keeping two different versions of the program. One of them is just the original program, the other is a copy of the original program in which all the query-goals and program predicates are renamed apart. It is this renamed version of the program that is analysed and optimized, and also the one that is started at the beginning. Although not stated in D.WP1.1M1, this renaming could affect the calls in meta-calls that are not *indeterminate*. In this way the *determinate* meta-calls will also use the optimized code. It is correct since, as said before, the corresponding predicate has been analyzed for this call and the optimizations have taken it into account. The renamed copy is analysed assuming that the only possible calls to each predicate are those that appear explicitly in the program (including meta-calls whose main functor is known). Thus, the program can be optimized according to this analysis information. Obviously, this renamed and optimized program cannot be used by the calls in meta-calls that were just variables at compile-time. For these calls the original version of the program will be used. Note that this will take place automatically because the terms that will be built at run-time will use the names of the original predicates. When we call a predicate in the original program, it will also call predicates in the original program. Thus, we guarantee correctness in the execution of the meta-call. Furthermore, as soon as the execution of the meta-call is completed, we continue executing the optimized program.

In conclusion, we have presented an automated way to analyse and optimize a program that contains variables in meta-calls. The method is transparent to the user, who does not need to supply any extra information. The drawback of this method is that we must keep two versions of the program, the original one and the optimized one. The relevance of the optimizations will depend on the *time* that we stay in each one

[1]Note, however, that if at run-time the meta-call it is still *indeterminate* an error will be reported.

of the versions. Execution will start in the optimized program and will move to the original program to compute a resolution subtree each time an *indeterminate* meta-call is executed. Then it will go back to the optimized program. As was also pointed out in D.WP1.1M1, in case space is a pressing issue, the user should be given the choice of turning this copying on and off.

A different approach is taken in Aquarius [5], in MA3 [6] and in previous versions of the PLAI analyser [2], where the user must provide all the different types of calls to predicates that can appear inside meta-calls and in the body of asserted clauses. However, only one version is generated for each predicate. This version will be more or less optimized depending on the accuracy of the information supplied by the user. If the types of calls that can appear in the meta-calls are very general, then nearly all the opportunities for optimization will be lost. It can also be very tedious for the user to give information for all the possible new calls. Note that our solution does not preclude using this one as well (and this is planed in the PLAI system).

4 Dynamic Predicates

There are a number of predicates in Prolog that affect the program itself by adding to or removing clauses from the program. These predicates are typically `assert`, `abolish` and `retract`. Predicates that can be affected by them must usually be declared as `dynamic` in modern Prolog implementations. For this kind of predicates it is difficult to have an accurate analysis (at compile-time) because their code can be modified at run-time. This problem was partially solved in D.WP1.1M1, where three different types of dynamic predicates were established:

`data`	if only facts are asserted
`memo`	if only logical consequences of the program itself are asserted
`call`	in other case

For the first two types the analysis (and optimization) can proceed as usual (except perhaps for analyses such as that of granularity, which are affected by program execution time) because the run-time modifications of the predicate do not modify its abstract success substitution and do not introduce new calls to other predicates. Thus, all the assumptions about the program behaviour are still correct and thus also all the optimizations. However, in D.WP1.1M1 we introduced a limitation in the sense that no optimization should take place in the presence of a dynamic predicate of the `call` type to keep the program correct. In this report we will improve on this approach in such a way that the program can still be optimized.

As said before, there is no general method to analyse in advance a predicate whose clauses can be modified at run-time. This poses two problems:

1. The abstract success substitution for the predicate can change during program execution.

2. The literals in the body of the new clauses can produce new different types of calls.

Both problems compromise analysis correctness. In fact, if the abstract success substitution is no longer correct, new types of calls may appear, thus producing the same effect as problem 2. If a new type of call to a predicate not been dealt with appears, the assumptions for the optimization of such predicate may not hold and the optimization may not be correct.

Note that the use of the `abolish` and `retract` predicates does not affect analysis correctness. If we remove some or all the clauses for a predicate, the abstract success substitution for the predicate will still be correct (and a safe approximation of the run-time substitutions) and so problem 1 will not appear. However, we can lose accuracy, as the abstract success substitution for the remaining clauses (if any) may be more particular than the one we had obtained in the previous analysis. Problem 2 will not appear either as no new clause is being introduced in the program. The `assert` predicate is much more problematic, since it does potentially affect correctness of the analysis and, thus, correctness of the optimizations performed using analysis information.

To overcome the first problem mentioned above in the case of `assert`, it is always possible to generate a correct (but inaccurate) analysis of any predicate using $\top$ (the most general abstract substitution) as abstract success substitution. We can use an "appropriate" $\top$ making use of the characteristics of the domain. For example, if we know that a variable is ground, it will continue being ground. This is to be done for all the dynamic predicates of type `call`.

The second problem is that `assert` can introduce new calls in the program to any predicate and with any substitution, the same problem that appeared with indeterminate meta-calls. The solution then is to treat literals in the body of a clause that has been *asserted* like meta-calls. This involves again having two copies of the program, with and without optimizations, as explained in section 3. There we saw that meta-calls directly used the original version due to the renaming mechanism. The same applies here. Whenever a clause for a dynamic is asserted, the literals in its body will use the original predicates. In this way correctness is always assured. The discussion regarding the relevance of the optimizations is the same as in section 3.

5 Improving Accuracy. The "Trust" Directive

In this addendum we introduce a new directive (`trust`) that did not appear in D.WP1.1M1. It is a predicate-level directive that can be used to provide the compiler with extra information about the success substitution for a predicate. The syntax is

```
:- trust(goal_pattern, [call_declaration],[success_declaration])
```

and can be read as follows: if a literal that corresponds to `goal_pattern` is analysed and all the declarations in `call_declaration` hold for the associated abstract call substitution then the `success_declaration` holds for the abstract success substitution.

`call_declaration` can be empty (it is then assumed to be true). In this way we can state properties that must always hold for the success substitution, no matter what the call substitution is. It is always advisable to have a declaration in this format in case the call substitution does not hold for any of the `call_declarations`. We leave at this point the questions of in which order `trust` directives are searched for, what to do when more than one `trust` can be applied to an abstract call substitution, etc. open.

Note that in no case the existence of a `trust` directive saves analysing the predicate. It is necessary to analyse it in order to have all the possible input values for all the predicates that appear in the body of the predicate. Otherwise the analysis would not be correct.

As both the `trust` information and that generated by the analyser must be correct, the intersection of them must also be correct. Thus, we first translate the `trust` declaration into an abstract substitution[2]. The intersection between abstract substitutions (whose domain usually has a lattice structure) is computed with the *GLB* (Greatest Lower Bound) operator, usually represented as $\sqcap$. Although this operation may not be implemented in abstract analysers that compute their fixpoint upwards (as in the case of PLAI), we believe that, in general, it is not difficult to implement. If we decide not to implement the $\sqcap$ operation then we can use the `trust` information directly as a success substitution. In this case the `trust` directive must be used with care because if the information supplied is more general than that obtained by the analyser, we are just losing accuracy.

The `trust` directive is very useful for dynamic predicates (and also predicates whose definition is not available at the time of analysis). As previously stated, as we do not know the clauses a dynamic predicate will have at run-time, the solution was to take $\top$. This can make the analysis very inaccurate. However, in most of cases the user knows how the dynamic predicates are going to behave and can give the analyser valuable information regarding the success substitutions for the predicate. In conclusion, we have a way of optimizing programs that contain `call` dynamic predicates without losing too much accuracy (opportunities for optimization) using the renaming transformation and the `trust` declaration.

6 The "Entry" Directive and Multiple Program Specialization

In section 10 of D.WP1.1M1 we stated that each module (file) must have a *module* declaration that states the list of predicates that were exported by the module (to other modules or to be used directly by the user). In query-driven analysis this information

[2]This translation must be already implemented to understand `entry` directives.

is used to know starting points for the analysis.

In D.WP1.1M1 we introduced also the `entry` directive. It was used to express information that is valid at the entry point of a given predicate. This information can be used by the analyser to increase accuracy. Whenever possible it would be interesting to provide entry declarations for the predicates that are exported (appear in the module declaration). Several `entry` directives may exist for a predicate. In that case the meaning of the directives is a disjunction of the entries, but a *closed* one. This means that if entry directives appear, they must cover all the possibilities. For example, if we have two entries for a predicate, one saying that X is ground and the other that X is a free variable, then X can never be a term such as `f(A)`, that is neither a ground term or a free variable.

In the context of multiple program specialization [7, 4], several versions may be generated for each predicate in the program. Each one of these versions receives a unique name in the multiply specialized program. These unique names are automatically generated. However, in order to keep the multiple specialization process transparent to the user, whenever more than one version is generated for a predicate which appears in the `module` declaration, the original name of the predicate is reserved for the version that must be used when we call the predicate from outside the program. If more than one `entry` declaration appears for a predicate and different versions are used for different entries, it is not possible to assign them all the original name of the predicate (unless we used time-consuming run-time tests to determine the version to use). To solve this problem we have allowed the `entry` directive to have one more argument. Here the user includes the name the corresponding specialized version should have. For example, in

```
:-entry(mmultiply(A,B,C),[g(A),g(B)],mmultiply_ground).
:-entry(mmultiply(A,B,C),[],mmultiply_any).
```

if these two entries originate different versions, one would be called `mmultiply_ground/3` and the other `mmultiply_any/3`.

It may also be the case that several entries are collapsed into one version. In this case the version will get the name of any of the entries. Also, in order to allow the user to call each version with the name provided, we will add to the multiply specialized program clauses of the type:

```
m_free(A,B):-
        m_any(A,B).
```

which provide appropriate handles.

If the user does want multiple specialization but also wants to avoid the renaming conflicts involved, he only has to supply a single `entry` directive that summarizes all the different cases. In this way even if there are several versions for the predicate, there will only be one exported version, and that one will keep the original name.

7 Conclusion

In this addendum to TR CLIP3/93.0, we have reviewed and improved some of the ideas previously presented therein. We have discussed some source-to-source transformation techniques for *pruning* predicates to deal with the issue of *remote* cuts, not addressed there. More importantly, we have presented some analysis techniques that allow overcoming the difficulties in analysing *full* Prolog programs. We have given a method that allows automatically analysing and optimizing programs containing indeterminate meta-calls and unrestricted dynamic predicates, without losing too much accuracy (with the use of the `trust` declaration). Finally, we have presented a way to deal with multiple program specialization by means of an improved `entry` declaration.

References

1. F. Bueno, M. Carro, D. Cabeza, F. Ballesteros, P. López García, M. García de la Banda, M. Hermenegildo, L. ómez, S. Prestwich, and Shan-When Yan. A Proposal for an Interchange Abstract Syntax for (Parallel) Prolog. Technical Report CLIP3/93.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 1993.

2. F. Bueno, M. García de la Banda, D. Cabeza, and M. Hermenegildo. The &–Prolog Compiler System — Automatic Parallelization Tools for LP. Technical Report CLIP5/93.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 1993.

3. M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

4. A. G. Puebla and M. Hermenegildo. An Implementation Technique for Multiple Program Specialization. Technical Report CLIP7/94.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 1994.

5. P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

6. R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699, Seattle, Washington, August 1988. MIT Press.

7. W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.